Using Human Computation in Dead-zone based 2D Pattern Matching

Kamil Awid[1,2], Loek Cleophas[1,3], and Bruce W. Watson[1,2]

[1] FASTAR Research Group, Department of Information Science
Stellenbosch University, Republic of South Africa
[2] Centre for Artificial Intelligence Research
CSIR Meraka Institute, Republic of South Africa
[3] Natural and Formal Languages Group, Department of Computer Science
Umeå University, Sweden
{kamil, loek, bruce}@fastar.org

Abstract. This paper examines the application of human computation (HC) to two-dimensional image pattern matching. The two main goals of our algorithm are to use *turks* as the processing units to perform an efficient pattern match attempt on a subsection of an image, and to divide the work using a version of *dead-zone* based pattern matching. In this approach, human computation presents an alternative to machine learning by outsourcing computationally difficult work to humans, while the dead-zone search offers an efficient search paradigm open to parallelization—making the combination a powerful approach for searching for patterns in two-dimensional images.

Keywords: image pattern matching, dead-zone pattern matching, human computation

1 Introduction, motivation, and related work

In human computation, humans are used for tasks for which humans are more suitable than computers, i.e. they are human processing units, typically called *turks* in this context. We consider the problem of utilizing turks in the search for an object inside of a matrix of objects. To be more precise, this paper examines the utilization of turks in the search for occurrences of an image (the *pattern image* or *pattern*) in a larger image (the *subject image* or *subject*). This presents a two-fold problem: 1) efficiently exploring and dividing the search space (i.e. the image); and 2) utilizing a turk to check whether the pattern actually occurs in a specific area of the image. The algorithm that is to be used for the search is an adaptation of a dead-zone based pattern matching algorithm [7]; the new algorithm generalizes from this in that it performs a search on a *two-dimensional* matrix instead of on a *one-dimensional* array of symbols. The use of human computation allows more powerful searching that otherwise may be very difficult to solve with pure computational algorithms since humans easily recognize images that have been rotated, scaled, sheared, images with alternative colours, and vague patterns.

1.1 Human Computation

Human computation (HC) provides a mechanism for solving problems using humans as an alternative to using concepts of machine learning (ML) and artificial intelligence (AI). Human computation relies on a series of so-called *turks* which juxtapose computer processing units for processing information. Using HC it is possible to solve a

Kamil Awid, Loek Cleophas, Bruce W. Watson: Using Human Computation in Dead-zone based 2D Pattern Matching, pp. 22–32.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

myriad of problems that are trivial for humans yet baffle sophisticated programs [4]. The majority of these problems lie outside of what ML and AI can currently solve. Captcha [11] and more generally the Amazon Turk [6] service both attempt to solve problems that would be otherwise difficult to solve computationally.

The goal of this paper is to provide an alternative rather than a replacement for machine learning in the development of a 2-dimensional search algorithm. The two concepts are orthogonal and can be used in conjunction with each other, however, that is outside the scope of this paper. Due to the multidisciplinary nature of human computation, a number of considerations have to be taken into account at various tiers including high level design, algorithms, and human-computer interaction and other human aspects.

The work in 2-dimensional image search can be applied to satellite imaging data where turks can classify objects on a map and perhaps even train machine learning models. Letting a single turk scan such imaging data (possibly gigabytes of imaging data) may be quite cumbersome and prevents parallelism and verification. This work in this paper is motivated by such examples. The aim is to use humans and the computer to distribute tasks based on their respective strengths efficiently.

Work in HC has been approached from multiple directions. Researchers from MIT CSAIL have developed a javascript library to deal with the intricacies of HC [8] enabling easy parallelism, crash-and-rerun programming, and ease of implementation. In case a human computation program encounters an error, it is important to recover since external calls are expensive (i.e. re-running the entire task with a turk may cost time or money); a crash-and-rerun program mitigates this problem by recording computationally expensive information and allowing the use of this information to rerun from the last known point. Well known functions for parallelism such as fork and join are available through this toolkit.

Luis Van Ahn dealt with problems in motivation (i.e. monetary motivation and game theory), interfaces, and algorithms in his PhD thesis [4]. Other research in HC includes task routing [1], combining human and machine intelligence [2], parallelization and design patterns [3]. Additionally, Amazon has developed an environment where the developers can utilize human computation through their Amazon Turk service [6]. While Amazon provides an interface for connecting with turks, consideration still has to be given to development of efficient and coherent algorithms optimized for processing by humans (i.e. turks).

HC search algorithms allow the matching flexibilities of a human, and therefore the inputs can range across many object types, including sounds, pictures, or strings. With such flexibility, however, the fuzziness of the output increases. This may have beneficial and adverse effects depending on the problem at hand. In the case of a search algorithm, the uncertainty revolves about incorrect pattern matching and not completing assigned tasks. These issues can be mitigated through parallelizing [9] the work through different turks and using voting [10] or statistical methods to decide whether the answer is correct. There are other ways to mitigate output errors by rephrasing the problem in a way that results in the capture of natural human instincts in solving a problem [5]. Parallelization, however, lets us measure the degree of confidence of the final result by taking a sample of human outputs. Additionally, parallelization reduces a dependency on a single turk, reducing the amount of time it takes to solve a problem e.g. if a particular turk is unavailable at the moment.

1.2 Dead-zone pattern matching

Dead-zone (DZ) pattern matching [7] is an approach for string pattern matching—finding all occurrences or matches of a pattern string p in a larger string or text S. In a nutshell, DZ algorithms start from a situation in which a single *live-zone*—the entire text S—exists, and select a pivot in such a live-zone. They then proceed in checking whether an actual match of p occurs there—if so, this match is reported. Based on the information gathered during this checking, the algorithm can *dead-zone* particular areas to the right and left of the pivot—preventing unnecessary further match attempts in these areas, and splitting the live-zone into two separate smaller ones. It repeatedly processes such a live-zone (or multiple ones in parallel), until a situation is reached where no live-zone remains, and all of the text is dead, with all pattern occurrences having been reported.

Each algorithm from the DZ family is easily parallelized and therefore especially useful in the field of human computation where parallelization is necessary to have the same tasks processed by multiple turks in order to deal with uncertainty.

2-dimensional pattern matching is about finding occurrences or *matches* of a 2-dimensional pattern p in a 2-dimensional symbol matrix S. Figure 1 is a representation of what a 2-dimensional search algorithm tries to accomplish. The dead-zone algorithm starts a pattern match attempt in the middle of S. If a match is not found, the algorithm proceeds to shift in 4 directions using the data obtained during the match attempt. Once the shifting is complete and the dead-zones have been determined, the algorithm divides the matrix into 8 areas and recurses into each zone.

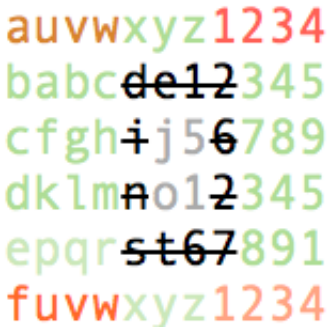

Figure 1. Symbol matrix **S** with occurrence of a 2x2 square pattern p (j5o1), dead-zone drawn around (struck-through text), and 8 areas created subsequently from the dead-zones (top, top-left, top-right, middle-left, etc.)

2 An algorithm for 2-dimensional dead-zone matching using human computation

Our new algorithm processes 2-dimensional data in the form of an image. The algorithm below matches an image contained in a larger image. Humans are best utilized in the processing of generalized problems i.e. problems that avoid detailed information, since the end result may be an approximation - while a turk may have a hard time recognizing pixels on the screen, he or she can certainly discern whether pictures made out of these pixels are similar. Therefore, the 2-dimensional data used for our algorithm is in the form of an image instead of other symbols. Nonetheless, the algorithm can be applied to any matrix of symbols.

The algorithm relies on dead-zoning a part of the image, then proceeding to shifting, dividing and delegating the remaining work to turks. This approach allows the smaller instances of the problem (smaller live-zones) to be easily parallelized between numerous turks.

The algorithm uses the TurKit algorithms developed at MIT CSAIL [8]. Namely, we are using crash-and-rerun concepts, fork (to allow parallel processing), createHIT (to create our task for the turk), and voting (determine whether the turks agree on the results). TurKit utilizes HTML to generate interfaces.

```
1
2  function human_2d_dz(live_low, live_high)
3  {
4    if(<pattern larger than livezone>)
5      return null;
6
7    draw_viewport(live_low, live_high);
8    pattern_found =human_search(); // Turk's work
9
10   if(pattern_found)
11   {
12     console.log("Match at " + live_low + " " + live_high);
13     fork(function(){
14       // vote on the correctness of the result
15       vote_result =mturk_vote(pattern_found, ..);
16     })
17
18   }
19
20
21   //Expand zone and let user estimate whether there is a possibility of the
         pattern occurring on any of the sides of the viewport.
22   //Let the user indicate how far they shift in the image to create deadzones.
23   new_deadzone =expand_deadzone(live_low, live_high);
24
25   live_zones =create_live_zones(new_deadzone)
26   for(zone in live_zones)
27   {
28     fork(function(){
29         human_2d_dz(
30           live_zones[zone].live_low,
31           live_zones[zone].live_high
32         )
33     });
34     join() ;
35   }
36
37 }
38
39 //Example of a human search function utilizing TurKit
40 function human_search()
41 {
42   var hitId =mturk.createHIT({
43          title : "Find possible image zone",
44          desc : "Indicate whether the pattern is found in the viewport.",
45          url : votePage,
46          height : 800,
```

```
47          reward : 0.01,
48    })
49
50    return mturk.waitForHIT(hitId).assignments[0].answer.found;
51  }
```

Listing 1.1. 2D DZ algorithm using TurKit

2.1 Pattern Match

The algorithm utilizes a viewport (Figure 2) to deal with a particular subsection of the image to be searched. The viewport is a rectangular region, just like the image. The procedure *draw_viewport* (Line 7) takes the entire image and the live zone (indicated by 2D coordinates *live_low* and *live_high*) and creates a viewport that is to be processed by *human_search* (Line 8). After processing by a turk, *human_search* returns a boolean flag to indicate whether a pattern occurrence was found.

Figure 2. Viewport displayed to the user.

The turk is initially presented with the viewport centred in the middle of the original image s. In the case that the turk finds the pattern p completely inside the viewport, the algorithm returns the position of the image inside s. In any case, the program continues to search for other occurrences of the pattern.

The viewport has size p to keep the result of the turk operations consistent with the problem: the pattern found must fit within the viewport i.e. patterns larger than the viewport will be ignored.

When the viewport completely overlays existing dead-zones, we can be confident that the pattern of size of p cannot be found inside this viewport. In the case of scaled patterns, we cannot confidently state that the pattern does not reside in a viewport partially obscured by a dead-zone, preventing the algorithm from skipping the area to be examined by the turk.

2.2 Expanding the dead-zone and shifting the viewport

The next step in the algorithm, *expand_deadzone* (Line 22), draws a zone around the viewport for the turk to indicate the dead-zones (Figure 3). The turk is then asked by

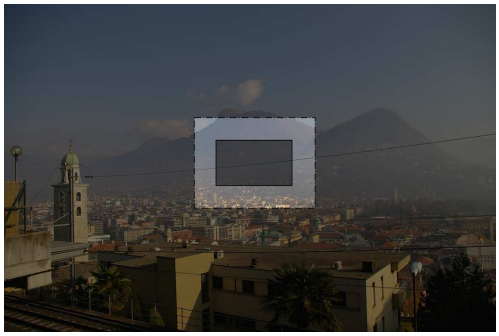

Figure 3. Dead-zones drawn around viewport.

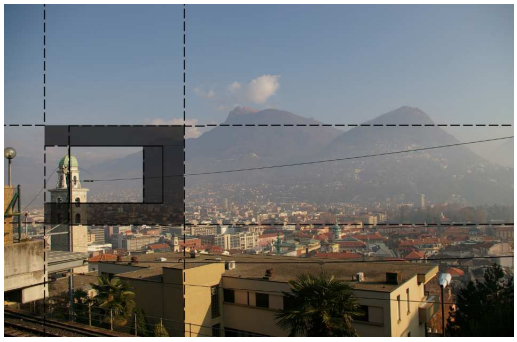

Figure 4. In the case that a partial match is found (ex. tower in the image), the dead-zones will be indicated, and the viewport will be shifted accordingly based on user distance indicated.

the algorithm to estimate the appearance of the pattern next to the current viewport. Once the turk indicates where the pattern could possibly occur, we can shift and infer new dead-zones.

In the case that a partial match is found (i.e. p is overlapping zones), the turk indicates the dead-zoned areas, and the algorithm moves the viewport based on the shift distance indicated. (Figure 4)

2.3 Pattern not found/Slicing

If the user has indicated that there is no possibility of the pattern in the extended dead-zone the algorithm proceeds to slice s into 8 zones starting from each corner (Figure 5) by using the procedure *create_live_zones* (Line 24) which returns the aforementioned 8-zones with live-zone data. The algorithm recurses into each of the rectangular zones created and performs all the steps outlined above until a pattern is found or all zones the size of p are dead-zoned. The algorithm utilizes parallelism introduced in TurKit. The TurKit *fork* function works on the same principles as the machine equivalent of the function i.e. it creates a new process. The function passed into the fork further divides the problem and eventually returns the result. The different return values are synchronized through the *join* function.

2.4 Completion

A critical step to completion of the algorithm is resolving issues with fuzziness. In the process above, fuzziness is mitigated by having the turks vote using the *mturk_vote* (Line 14) function from the TurKit library. Alternatively, taking multiple samples of the results processed by turks and performing regression analysis is possible (not shown above).

3 Expected Case

While there are no experimental results at this stage, we can reason about the expected case of the algorithm. Figure 6 presents a typical search case in the algorithm using 3 turks as an example. Parallelization of the task has not been shown, however,

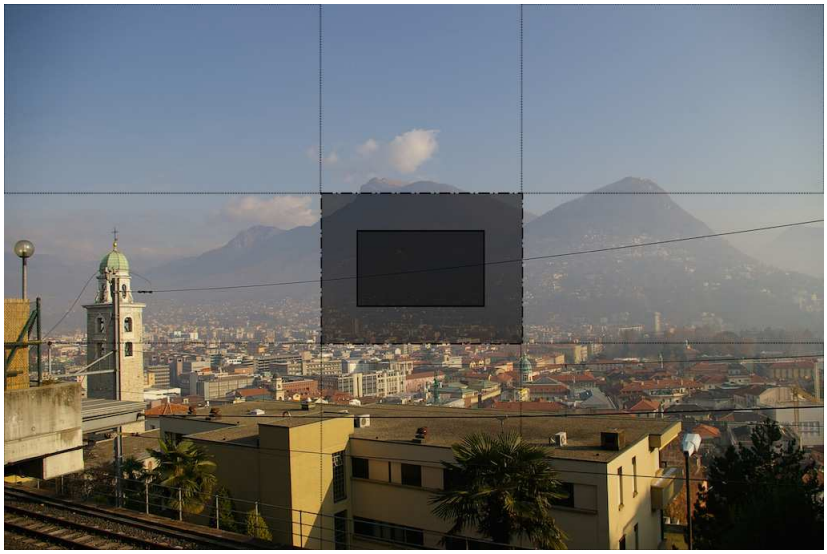

Figure 5. New zones to be analyzed.

it is feasible to send the initial task to all three turks starting at different locations of the viewport.

At the start of the algorithm, the viewport is generated and, along with the pattern, sent to Turk 1. Turk 1 indicates that the pattern is not in the viewport, and not likely to be near i.e. the dead-zones are inferred. The algorithm will make a number of shifts from areas without possible matches when the dead-zone is indicated by the user. This yields an advantage over scanning areas sequentially since the algorithm does not have to check every area of S.

Subsequently, the algorithm then slices an image into new live-zones and sends the data to Turk 2 which indicates the pattern is near. The algorithm then shifts an amount indicated by the user and sends new live-zones to Turk 3 where a match is finally indicated.

4 Robustness

As stated before, due to the parallelized nature of the algorithm we are able to reduce the error in computing problems. In the case of the DZ algorithm, chunks of the images can be sent to multiple turks for verification. Parallelized results are then compared to verify with a lower degree of error that the final result is valid. This mitigates mistakes and inherent change blindness in the turk. In the current version of the algorithm, voting is used to verify results.

The development of a natural algorithm heavily relies on concepts from the field of Human Computer Interfaces. A search algorithm must consider human memory principles for processing data i.e. the short term memory is limited to seven chunks at a given time while the long term memory is useful for seeing larger patterns [12]. Chunking is designed to deal with human memory limitations, namely limiting the number of artifacts on the screen by splitting data into meaningful pieces (a well known example of that are phone numbers where the area code is separated from the rest of the number). Chunking the information for the search algorithm can decrease the processing time and reduce errors in turk processing. The algorithm

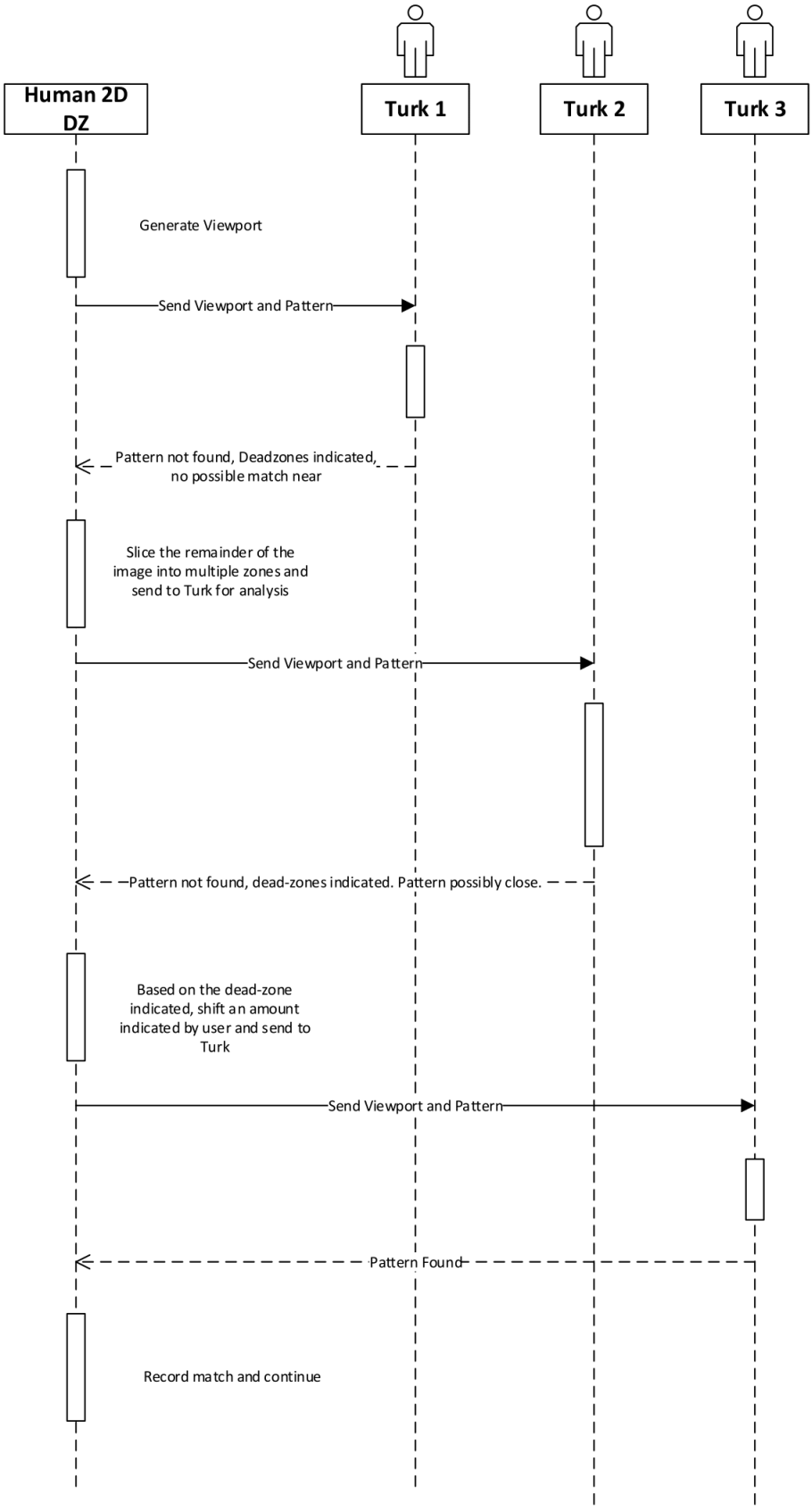

Figure 6. Sequence of Human 2D DZ.

utilizes chunking to help the user examine smaller sections of data to find a pattern match.

A problem the algorithm runs into is the definition of problem bounds. If the turk is asked to detect an object on a 2D surface, the turks may bring their own biases. An example of such a bias could be an object that is differently coloured or sized, but otherwise the same. Limiting the scope of the problem is a critical part of the algorithm. The algorithm does not deal with this problem, it is left for the user to provide a sufficient scope to the turks.

5 Performance

The greatest cost savings this algorithm provides is in dead-zones indicated by each turk. The entire image does not have to be scanned if there is no chance for a match.

The performance of the algorithm largely varies from turk to turk; for example adults may take more care in scanning the viewport while children may haphazardly scan the same area. Research outlined in [8] has shown a number of results. The experiments considered a number of problems dealing with blurry text, iterative writing and photo sorting. Part of the time is spent on waiting for turks to accept tasks and waiting for turks to perform the work, which is to be expected from a natural system. As mentioned before, effective chunking increases the performance of the algorithm, but there may be cases where chunking is not possible. Additionally, the time to compute with a turk will be significantly different than using a machine. For this reason, it is necessary to separate "human time" from "computing time". For example, the time to slice and shift is largely dependent on computing resources, while the time to find a pattern is dependent on the turk recognizing the object. For this reason we introduce $O(ht)$ for the asymptotic notation of human time. The 2D algorithm's time complexity would therefore be $T(n) = 8T(\frac{n}{8}) + ht \times n$, where the work done outside the recursion is indicating dead-zones and shifting. In the case of this algorithm, the running time will depend on the marking of the potential dead-zone and shift performed by the user. The worst case for machine computation, where major shifts do not occur, being $O(n \log n)$.

The dead-zone algorithm has a number of advantages for searching text. In the case of most algorithms, the worst case scenario is quadratic $O(|S|^2)$ while the best case scenario is $O(\frac{|S|}{|p|})$. In the case of a DZ algorithm, the worst case scenario remains the same, however, the best case scenario is significantly improved. The best case scenario yielded by the DZ algorithm is $O(\frac{|S|-|p|+1}{2|p|-1})$. In practice the improvement is significant since the algorithm performs half the match attempts. Additionally, the algorithm is easily parallelisable which is a key in battling the latency presented by human computation.

A large part of the performance will depend on the Human-Computer Interfacing due to the high amount of interactions and latency between the turk and the machine. A well designed interface will make the process seamless by removing obstructions for new turks in the process. As mentioned above, the time it takes the human to perform the task will be vastly different from the time it takes a machine to perform the same task. In the case of a human, and additionally, the time to process will differ from turk to turk. A well designed interface will optimize the processing and reduce the average time spent by the turk. The experimentation which remains to be done will test this algorithm against other alternatives in the paper to gauge which performs

fastest in a real life scenario. The alternatives will include classic divide and conquer search algorithms and displaying the entire matrix to the user for their peruse.

Furthermore, the cost of running turks will have to be tweaked and tested to determine the best balance in cost to difficulty of task ratio.

5.1 Parallelization

In order to create a robust algorithm the final result validity has to be measured to be reasonably accurate within a confidence interval. Parallelization occurs in two levels. The first level poses the problem multiple times to an array of turks. The second level parallelizes the work that needs to be completed in a single run by multiple turks. The chunking of the pieces to be found yields work that can be performed by multiple turks at the same time, asynchronously. This paves way for distributed human computation.

6 Conclusions and future work

Human computation is just beginning to scratch the surface with the introduction of such applications as Captcha and Amazon Turk. A human computation DZ algorithm can be used in various fields dealing with imaging. Some examples discussed before were dealing with pictures and sounds, however, more concrete examples of such pattern matching could include geotagging locations (human turks indicate where various locations on a picture are), screening for cancers (determining cancerous patterns on a photo) etc.

With the proven efficiency of a 1-dimensional DZ algorithm we are expecting a more efficient matrix search using the pattern recognition of a human turk. Additionally, using a human turk gives us the possibility of performing flexible image processing while keeping the cost and time of the turk down.

While most modern algorithms tend to examine machine learning and artificial intelligence, human computation departs from this concept by utilizing human turks to perform simple work in order to solve a bigger problem. Humans are currently inherently better at recognizing patterns and with continued expansion of social networks we are given more access to resources. Utilizing the power of distributed networks, human computation can lead to results faster with the help of traditional algorithms.

The next steps in the algorithm is to measure the running time and cost. Due to approximate nature of human computation, a sample of data comparing the two algorithms above will be taken. The data will measure the number of steps that are taken to find the needle in a haystack in order to get a more accurate cost. Additionally, the experiment will measure the time taken to find and the amount of false positives and negatives yielded by both algorithms. The goal of the algorithms is to optimize robustness, running time and user experience. Furthermore, human computation sorting and classification algorithms need to be examined and expanded. Humans have tendencies and biases, and therefore it is important to adapt algorithms to work more naturally with a human.

References

1. Haoqi Zhang, Eric Horvitz, Yiling Chen, and David C. Parkes: Task Routing for Prediction Tasks. Proceedings AAMAS 2012, pp. 889–896.
2. Ece Kamar, Severin Hacker, and Eric Horvitz: Combining Human and Machine Intelligence in Large-scale Crowdsourcing. Proceedings AAMAS 2012, pp. 467-474.
3. Greg Little: Programming with Human Computation. MIT, 2007.
4. Luis von Ahn: Human computation. Carnegie Mellon University, 2nd edition, 2005.
5. Daniel Villatoro, Jordi Sabater-Mir, Jaime Simo Sichman: Validation of Agent-Based Simulation through Human Computation: An Example of Crowd Simulation. School of Computer Engineering, Nanyang Technological University, Singapore 2012.
6. Amazon MTurk: Amazon MTurk FAQ. https://www.mturk.com/mturk/help?helpPage=overview
7. Bruce W. Watson, Derrick G. Kourie, and Tinus Strauss: A Sequential Recursive Implementation of Dead-Zone Single Keyword Pattern Matching. IWOCA 2012, LNCS 7643, Springer-Verlag 2012, pp. 236–248.
8. Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller: TurKit: Human Computation Algorithms on Mechanical Turk. Proceedings UIST 2010, pp. 57–66.
9. Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller: Exploring Iterative and Parallel Human Computation Processes. Proceedings HCOMP 2010, pp. 68–76.
10. Andrew Mao, Ariel D. Procaccia, and Yiling Chen: Better Human Computation Through Principled Voting. Proceedings AAAI Conference on Artificial Intelligence 2013.
11. Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford: CAPTCHA: Using Hard AI Problems For Security. Eurocrypt 2003, LNCS 2656, pp. 294–311.
12. George A. Miller: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. The Psychological Review vol. 63, no. 2, 1959, pp. 81–97.